Modular Language Composition for the Masses

Manuel Leduc, Thomas Degueule, Benoit Combemale

Modular Language Composition for the Masses

Manuel Leduc
Univ. Rennes, Inria, CNRS, IRISA
Rennes, France
manuel.leduc@irisa.fr

Thomas Degueule
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
thomas.degueule@cwi.nl

Benoit Combemale
Univ. Toulouse, IRIT & Inria
Toulouse, France
benoit.combemale@irit.fr

Abstract

The goal of modular language development is to enable the definition of new languages as assemblies of pre-existing ones. Recent approaches in this area are plentiful but usually suffer from two main problems: either they do not support modular language composition both at the specification *and* implementation levels, or they require advanced knowledge of specific paradigms which hampers wide adoption in the industry. In this paper, we introduce a *non-intrusive* approach to *modular* development of *language concerns* with well-defined interfaces that can be composed modularly at the specification and implementation levels. We present an implementation of our approach atop the Eclipse Modeling Framework, namely Alex—an object-oriented meta-language for semantics definition and language composition. We evaluate Alex in the development of a new DSL for IoT systems modeling resulting from the composition of three independently defined languages (UML activity diagrams, Lua, and the OMG Interface Description Language). We evaluate the effort required to implement and compose these languages using Alex with regards to similar approaches of the literature.

CCS Concepts • **Software and its engineering → Domain specific languages**; *Reusability*;

Keywords language concern, language composition, language interface, modular language development

1 Introduction

As recently demonstrated in the context of programming languages [4] and modeling languages [25], many software languages, including Domain-Specific Languages (DSLs), have a lot to share, e.g., recurrent constructs and paradigms. As "software languages are software too" [12], such recurrent pieces of software language specification and implementation would benefit from being developed separately to be eventually reused in new contexts.

The promise of modular language development is to liberate language designers from the burden of developing every new language from scratch and enable them to reuse, assemble, and customize existing *language concerns* to ease the definition of new ones [7]. Recent approaches in the area of modular language development are plentiful. Dedicated paradigms and underlying implementations have been explored to bring specific properties in language specifications, e.g., formal composability [4], or off-the-shelf specification in Spoofax [16], Monticore [17], and Neverlang [27]. However, the specific knowledge required to manipulate the corresponding paradigms hampers their wide adoption in the industry (i.e., *for the masses*). Other approaches, such as Lisa [22] or Melange [9], provide language reuse capabilities within frameworks and ecosystems relying on mainstream language engineering technologies (e.g., the Eclipse Modeling Framework which uses well-known object-oriented programming concepts [26]) but currently fail to support modularity at the language implementation level (e.g., the set of Java classes generated from an Ecore metamodel) which prevents opportunistic reuse of existing languages.

In this paper, we present a *non-intrusive* approach to *modular* language development that (i) can easily be integrated into mainstream (object-oriented) language engineering technologies, and (ii) is fully modular at the specification and implementation levels of language concerns.

At the specification level, language concerns expose clear interfaces that foster abstraction and information hiding [10]. The interface of a language concern expresses its requirements towards other concerns and encapsulates the internals of its implementation. Interfaces do not make any assumption on the internals of the syntax and semantics of required constructs.

At the implementation level, we present an object-oriented pattern supporting the composition of language concerns. It supports separate type-checking and compilation, and can be automatically generated from language specifications. This pattern leverages widespread practices in language engineering [26] and previous results on the application of object algebras [8] to modular language extensibility [19].

Concern composition most often requires glue, for instance to express how the evaluation contexts of two independent interpreters interact. Using concern interfaces, this glue is written *in terms of the interfaces only*. Besides, the glue between two concerns is expressed as a language concern itself, meaning that the knowledge required to compose two concerns is the same as that required to create a concern. Finally, two concerns can be substituted one another provided that they match the same interface.

We provide an implementation of our approach on top of the Eclipse Modeling Framework (EMF), using Ecore as the meta-language for defining the abstract syntax and ALE [19] as the meta-language for modularly defining the operational

semantics over Ecore meta-models (e.g., in the form of an interpreter). We extend ALE into Alex to include composition operators and an associated compiler seamlessly integrated within EMF that generates modular concern implementations conforming to the pattern we propose. We use Alex to re-implement a DSL for Internet of Things systems modeling and simulation that was used to evaluate Melange in earlier work [9]. We show that Alex supports a wide range of language composition scenarios, is intuitive to use, and supports modularity at the implementation level.

The remainder of this paper is organized as follows. Section 2 provides some background on modular language development and introduces a motivating example from which we derive a list of requirements. Section 3 gives an overview of our approach to modular language development and Section 4 presents the underlying implementation pattern we propose. In Section 5, we present Alex, a prototype implementation of our approach within the EMF ecosystem, and evaluate it in Section 6 on a use case consisting in the definition of a new DSL for IoT systems modeling. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Motivating Example and Requirements

In this section, we first introduce a motivating example in Section 2.1 which we use in the remainder of this paper. Then, we discuss in more details the notion of language interface in Section 2.2. Finally, we derive in Section 2.3 a list of requirements for modular language development that drive our approach.

2.1 Motivating Example

Let us consider the motivating example given in Figure 1 depicting the metamodel of a simple Finite-State Machine (FSM) language concern. It consists of a machine that contains a number of states and transitions between these states. States may declare local variables of arbitrary types. Transitions are guarded by a guard expression and execute an action. For convenience, the evaluation functions defining the operational semantics of domain constructs are depicted as method signatures in the corresponding meta-classes.

From the language designer's point of view, many expression languages would be good candidates for expressing guards, and many action languages would be good candidates for expressing actions. Rather than defining new guard and action languages from scratch, including their syntax, semantics, and tooling, it would be handy to reuse and plug existing languages that provide these functionalities into the base FSM language—OCL [5] for guards and Xbase [11] for actions, for instance. This would allow modelers to build FSM models by combining the expressiveness of the base FSM language with the expressiveness of dedicated expression and action languages, including their tool support.

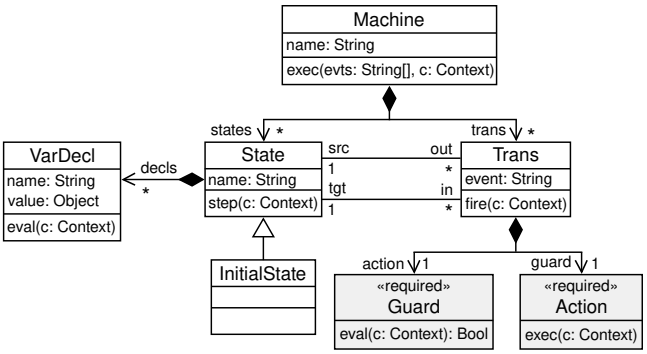

Figure 1. An FSM language concern with explicit interfaces

The question that naturally arises is: how to express the *required interface* of the FSM language? The notion of language interface in general is the subject of ongoing research [10]. In this paper, we are specifically interested in the interfaces necessary for language composition. From the FSM's standpoint, a guard is merely "an expression whose evaluation returns a Boolean;" that is, the signature of its evaluation function is $eval : Ctx \rightarrow Bool$. The internals of the expression language employed, e.g., its syntax (the set of Boolean operators it offers) and semantics (how they are evaluated) do not matter. In Figure 1, the `Guard` and `Action` constructs annotated with `«required»` denote the required interface expected by the base FSM language. The key idea here is that most of the language's semantics can be implemented independently from the syntax and semantics of guards and actions. Knowing that actions and guards can be evaluated with their respective evaluation function is sufficient to express the semantics of transitions—only the *signature* of their evaluation function is needed. In pseudocode, the semantics of the `fire` evaluation function of transitions could be written as follows:

```
fire(Trans t, Context ctx) {
  if (t.guard.eval(ctx))
    t.action.exec(ctx);
}
```

An interpreter for the evaluation semantics of the FSM language can be type-checked and compiled independently; but to be run, it needs concrete implementations of the execution functions of guards and actions. These will be provided later by other language concerns at composition time.

2.2 On Language Concern Interfaces

A language concern interface should expose the information needed to (i) use and (ii) compose a concern [7]. Using a language first involves producing a conforming model. The structural information, in the form of a metamodel, must thus be part of the concern's interface. Then, executions functions are invoked on the different model elements. The

set of execution functions, linked to the corresponding domain constructs, must also be part of the concern's interface. In contrast, details of the syntax of the required constructs, as well as the implementation of their execution functions can be encapsulated behind the interface, and it should not be necessary to inspect them in order to use or compose a language concern.

Another design choice to be considered by the language designer is the boundaries of a concern. While we do not enforce strict rules for the definition of language concern boundaries, we suggest following the well-known modularity principle of package cohesion which, in this context, states that (i) domain constructs that are commonly used together should belong to the same concern, and (ii) a concern should not have more than one reason to change. For instance, when designing the FSM concern of Figure 1, the language designer may wonder whether guards should be included as part of the language concern or as part of its required interface. The `Guard` construct is clearly needed, but its concrete realization is subject to discussion. First, the implementation of the various constructs enabling the expression of guards is complex and will probably overtake the complexity of the rest of the concern if it would be included. Besides, there are already existing expression languages that could fulfill this functionality and are evolving at their own pace, independently from the FSM concern. Hence, the `Guard` construct is defined as `«required»` and is expected to be provided later by another language concern through composition.

2.3 Language Composition Requirements

From the introduction, motivating example, and the notion of language concern interface, we derive a list of five requirements that must be addressed for non-intrusive and modular language development. Following the terminology introduced for concern-oriented language development [7], we refer to languages as language concerns, regardless of whether they expose an explicit required interface or not.

Concern Encapsulation (R1) Language concerns should be composed without having to inspect their internal implementation. In other words, the information exposed in the interfaces of language concerns should be sufficient to enable type-safe composition of language concerns.

Explicit Required Interfaces (R2) Required interfaces of language concerns should explicitly state the requirements a concern has towards other concerns. Knowing the interfaces only should be sufficient to state on the validity of the composition of different concerns. A composition is valid if the requirements expressed by the required interface of a concern are fulfilled by other concerns, and if it can be ensured that the generated implementation will compile. Checking the validity of the composition of concerns should be possible at the level of the meta-language used for the definition of language concerns, without requiring code generation.

Separate Compilation (R3) Language concerns should be type-checked and compiled separately, and should not have to be edited or recompiled to be composed with other language concerns. Furthermore, language concerns should not make any assumption on the way they will be reused and composed, i.e., they should not *anticipate* reuse.

Concern Substitutability (R4) Two language concerns providing constructs that match the same interface should be substitutable one another in the context where this interface is required. From the requiring concern's point of view, the choice of a particular language concern should be transparent. Substitution of a language concern by another should not require any modification of the language concern which depends on it.

Non-intrusivity (R5) The definition of language concerns satisfying the requirements above should not disrupt widespread language engineering processes, such as abstract syntax definition in the form of object-oriented metamodels. It should not require a new paradigm for the specification of language concerns to be broadly applicable into mainstream language engineering technologies and to foster its adoption.

3 Approach Overview

Figure 2 gives a high-level overview of our approach and is discussed in more detail below.

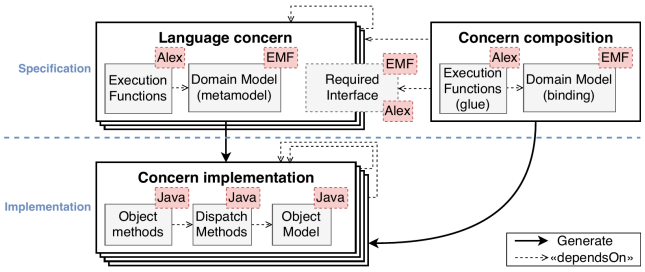

Figure 2. At the specification level, language concerns, their interfaces, and the composition between them are expressed in EMF and Alex. A dedicated compiler generates fully modular Java implementations from both the concerns themselves and the specification of their composition

At the specification level, a language concern is expressed following a standard metamodeling process. On the one hand, its abstract syntax is specified by an object-oriented metamodel, i.e., a set of meta-classes corresponding to domain concepts including their properties and the relations between them. On the other hand, its operational semantics is defined by a set of execution functions woven on corresponding constructs of the metamodel [9]. In the case where execution functions have to manipulate run-time data, these would be specified in a separate dedicated metamodel.

Many formalisms could be used to express these two aspects. In our approach, we use Ecore [26] to write metamodels and an extension of ALE [19], named Alex, to express the operational semantics. Following the philosophy of ALE, Alex is a simple action language based on Xbase [11] that enables weaving execution functions in Ecore classes using static introduction [23].

As mentioned in Section 2, language concerns may expose a required interface that materializes their requirements towards other concerns. To express these interfaces, we rely on the built-in annotation mechanism of Ecore to enable language designers to add a «required» annotation on classes of the metamodel that constitute the required interface. The execution functions woven on such constructs consists of signatures only: they express what semantics is expected from the other concerns that will fulfill these requirements.

The very same meta-languages are used to express how to compose two concerns. To specify that a required construct is realized by an external construct in another concern, we employ a simple delegation pattern between the two constructs: the required meta-class is extended by a new meta-class, in a new metamodel, that holds a reference towards the external construct. A new Ecore metamodel is created to bind all constructs of the required interface of a concern in this way. The glue between the signatures of the execution functions of required constructs and the implementation of these execution functions in another concern is expressed in Alex itself. The meta-class that holds the delegate reference implements the required signature in Alex; its body expresses how to glue together the two concerns semantically. Concretely, this means that the skills required to define new concerns are the very same as those required to express how to compose these concerns.

Language concerns are compiled to Java code using two separate compilers: the built-in Java compiler of EMF that compiles Ecore metamodels to a set of Java interfaces and classes, and our own compiler of the Alex meta-language that generates a set of Java interfaces following the pattern introduced in Section 4. Language concerns can be type-checked and compiled independently of each other. The very same compilation chain is reused to compile the specification of the composition of two language concerns. From a description of the bindings between two concerns in the form of an Ecore metamodel and the glue between their semantics in the form of Alex execution functions, a separate set of Java interfaces that composes the two concerns is generated.

In the next section, we dive into the implementation pattern itself and highlight how it enables modular composition of such concerns.

4 Modular Language Implementation

In this section, we describe how to derive modular language concern implementations from specifications of language concerns as described in Section 3. We use Java as the object language to detail the pattern and its properties as it is the language of choice in the EMF ecosystem. It can however be adapted effortlessly to any mainstream object-oriented programming language that supports (i) parametric polymorphism (generics) with bounded type parameters, (ii) multiple class or interface inheritance, and (iii) single dynamic dispatch. For instance, it is trivial to implement the same pattern in Scala using traits or in C++ using multiple class inheritance and templates.

The modular implementation pattern we propose relies on two main ideas that make it intuitive. First, it leverages two well-known concepts of object-oriented programming: inheritance and the delegation pattern. Second, the same pattern and compilation scheme are employed to implement both the language concerns themselves, and the specification of the composition between them.

Our pattern extends the Revisitor pattern that was used in earlier work to support modular and independent extension of the syntax and semantics of DSLs [19]. In this section, we describe how we extend it to account for required interfaces and go beyond strict extension to support arbitrary composition of language concerns. Our extensions retain the desirable properties of the Revisitor pattern: the syntax and semantics of language concerns can be independently extended in a modular and type-safe way, without requiring anticipation.

4.1 Language Concern Implementation

In our approach, the abstract syntax of language concerns is defined by an object-oriented metamodel such as the Meta-Object Facility (MOF) [24]. In particular, our implementation uses Ecore metamodels. The native compilation chain of EMF automatically generates a set of Java interfaces and corresponding implementation classes for every meta-class in an Ecore metamodel [26]. As we have shown in earlier work, it is possible to automatically generate a Revisitor interface from the same metamodel, which specifies an extensible mapping from syntactic objects to semantic objects, captured by generic type parameters of the interface [19]. In the following, we recall the main concepts of the Revisitor pattern when necessary—the interested reader may refer to our previous work for a complete overview.

We extend the Revisitor implementation pattern to account for «required» constructs. We enable language designers to annotate certain elements of the metamodel with a «required» annotation, using the native EMF annotation mechanism [26]. «required» classes must be declared abstract, as they cannot be instantiated in the current language concern without being bound first. Annotating a class with «required» is a simple language concern interface documentation which is both understandable by humans, who can quickly understand if a language is fully defined and what are its extension points, and by computers which can exploit

```
interface GFSMRev<M, T, A, ...> {
  M machine(Machine it);
  T trans(Trans it);
  // No factory method for Action
  ...

  default M $(Machine it) { return machine(it); }
  default T $(Trans it) { return trans(it); }
  A $(Action it); // Abstract dispatch
  ...
}
```

Listing 1. Revisitor interface for the FSM language concern depicted in Figure 1

them to check interfaces and bindings at the specification level automatically.

A first part of the pattern, the *semantic mapping*, specifies a mapping from meta-classes to abstract execution functions and is implemented by a Revisitor interface. In a standard Revisitor interface, each meta-class leads to the introduction of (i) an abstract factory method and (ii) a dispatch method that dynamically dispatches from static metamodel types to the appropriate factory method according to the run-time type of the argument. As the «required» classes are not meant to be fully implemented in the current concern, the generated Revisitor includes an (abstract) dispatch method but skips the generation of a factory method for the «required» classes. The generation of abstract factory methods is postponed until concrete implementations of «required» classes are known, i.e., until the requirements expressed by «required» classes are fulfilled by one or several other concerns at composition time.

Listing 1 depicts an excerpt of the Revisitor interface generated from the FSM metamodel of Figure 1.[1] The GFSMRev Revisitor interface declares one generic type parameter per class in the metamodel (including the «required» ones) and one factory method per non-«required» class. Consequently, there is no factory method for Action and Guard. Finally, the Revisitor interface declares one dispatch method (named $) per class in the metamodel. The $-methods define a case-based mapping from syntactic constructs to corresponding semantic objects, where the mapping is executed lazily, and explicitly, through invocations of the $-methods. As concrete implementations of Action are not known yet, its dispatch method is left abstract.

A second part of the pattern, the *semantic interface* is realized by a set of Java interfaces—one per meta-class in the concern—that define the signatures of the execution functions of the constructs included in a concern. The signatures are then mapped to the appropriate constructs through the definition of a *concrete semantic mapping*, a Java interface

[1] In the listings, ... depicts peripheral code left out for the sake of clarity.

```
interface IPrint { String print(); }
interface PrintGFSMRev
  extends GFSMRev<IPrint, IPrint, IPrint, ...> {}
```

Listing 2. Semantic interface and semantic mapping for a pretty-printer of the FSM language concern depicted in Figure 1

```
interface ImplPrintGFSMRev extends PrintGFSMRev {
  default IPrint machine(Machine it) {
    return () →
      "machine " + it.name + "\n" +
      it.states.stream().map(s → $(s).print()) + "\n"
      it.trans.stream().map(t → $(t).print());
  }
  default IPrint trans(Trans it) {
    return () → it.event +
                "[" + $(it.guard).print() + "]" +
                " / " + $(it.action).print();
  }
  ...
}
```

Listing 3. Implementation of a pretty-printer for the FSM language depicted in Figure 1

that inherits from the Revisitor interface and binds every generic type parameter to the corresponding Java interface.

Listing 2 presents a pretty-printing semantic interface for the FSM of Figure 1. The IPrint Java interface defines the signature of a print() method which returns a String. The PrintGFSMRev interface inherits from GFSMRev and binds each of its generic type parameters to the IPrint interface. So, every construct of the concern, as defined in its metamodel, are mapped to the print() execution function through invocations of the $-methods. At this point, the whole public interface of the language concern is defined, without any concrete implementation of the execution functions yet.

Finally, the *semantic implementation* is realized by a Java interface that inherits from the concrete semantic mapping and implements the factory methods to provide the implementation of execution functions. Each implementation is realized by returning instances of the semantic interfaces corresponding to the bindings defined in the semantic mapping.

Listing 3 depicts the pretty-printing semantic implementation for the FSM concern of Figure 1. A new ImplPrintGFSMRev interface is defined, extending PrintGFSMRev with concrete implementations of the factory methods using anonymous classes that give the semantics of every non-«required» construct.

It is important to note that the semantics of every non-«required» construct can already be implemented, even though the concrete syntax and semantics of «required» constructs

are not known yet. For instance, in Listing 3, printing a transition consists in printing its event, the associated guard, and the associated action. Invoking the $-methods on `Guard` and `Action` returns the semantic interfaces of guards and transitions, which have been mapped to `IPrint` in Listing 2 and allow to type-check and compile the FSM concern independently. The concrete implementations of `print()` for `Guard` and `Action` are not known yet, however, so `ImplPrintGFSMRev` cannot be instantiated and executed yet.

4.2 Composition of Language Concerns

As mentioned earlier, the composition of two language concerns is realized by a language concern itself. In this scenario, the metamodel of this new concern binds the «`required`» constructs of a requiring concern to the concrete constructs of one or several providing concern. The operational semantics of this new concern specifies how the signatures of the execution functions of the «`required`» constructs are bound to concrete execution functions of the providing concerns, possibly with some glue in-between. In the following, Section 4.2.1 details syntactic bindings between constructs and Section 4.2.2 details semantic gluing between execution functions.

4.2.1 Metamodel Composition

The first step in composing two language concerns is to compose the metamodels defining their abstract syntax. Composing the metamodels can be done regardless of the semantics of the composed language concerns, i.e., two language concerns can be mapped syntactically without having to consider their semantics.

Metamodel composition is realized by reusing the built-in inheritance mechanism of Ecore and the syntactic extension mechanism provided by the Revisitor implementation pattern. A new metamodel is created containing one *Bind* meta-class per «`required`» construct. Each *Bind* meta-class inherits from a «`required`» construct and holds a single reference `delegate` to the construct that fulfills the interface in the providing concern, following the well-known object-oriented delegation pattern. This way, the binding mechanism is non-intrusive and does not require any modification in either of the two composed concerns.

Figure 3 illustrates the composition of the FSM language concern to an Action Language concern and an Expression Language concern. The `FullFSM` metamodel specifies the bindings between constructs of the three concerns. Two bindings are defined for the «`required`» classes `Action` and `Guard`, respectively to the `Block` and `Exp` classes, materialized by the `BindAction` and `BindGuard` meta-classes. Following the same generation process as introduced in Section 4.1, this leads to the generation of a new Revisitor interface which inherits from the Revisitor interfaces of all composed language concerns and specifies factory and dispatch methods for the *Bind* meta-classes. The `FullFSMRev` interface depicted in Listing 4 is the Revisitor interface generated from the `FullFSM` metamodel shown in Figure 3. As the concrete types of `Action` and `Guard` are now known, the `FullFSMRev` gives concrete implementations for the dispatch methods that were left open in Listing 1.

```
interface FullFSMRev< ... , ActionT, ... ,
                      BindActionT extends ActionT, ... >
  extends FSMRev< ... , ActionT, ... >,
          ALRev< ... >,
          ExpRev< ... > {
  BindActionT bindAction(BindAction it);
  ...
  default ActionT $(Action it) {
    return bindAction((BindAction) it);
  }
  default BindActionT $(BindAction it) {
   return bindAction(it);
  }
  ...
}
```

Listing 4. Revisitor interface generated from the `FullFSM` metamodel depicted in Figure 3

4.2.2 Semantic Interface Composition

Once «`required`» constructs are bound to concrete constructs syntactically, their semantics must be bridged. In the FSM example, the three concerns support variable binding and manipulation. As they have been defined independently, however, the stores that hold variables and their values do not match: the Action Language concern uses the `al.Env` store which holds integer variables, the Expression Language concern uses the `exp.Env` store which holds Boolean variables, and the FSM language concern uses its own `fsm.Ctx`. It is nonetheless essential that the variables declared in the FSM can be manipulated by guards and actions. In the FSM example, the semantic glue thus mainly consists of the translation of variables from one store to the other and the invocation of the appropriate execution functions.

The two boxes at the bottom of Figure 3 depict a possible glue between these concerns. As shown in the bottom left, executing an action requires to extract the integer variables declared by the FSM from the `fsm.Ctx` store and to provide them to the actions using their own `al.Env` store. Then, invoking the `eval()` execution function of `Block` is done through the `delegate` reference hold by `BindAction`, passing the appropriate store. Finally, the local `fsm.Ctx` store of the FSM is updated back to account for possible updates of the values by the actions. As shown in the bottom right, a similar glue is defined between `Guard` and `Exp`, this time passing the Boolean variables around.

Listing 5 depicts how the glue is implemented following our implementation pattern. The glue is specified as the

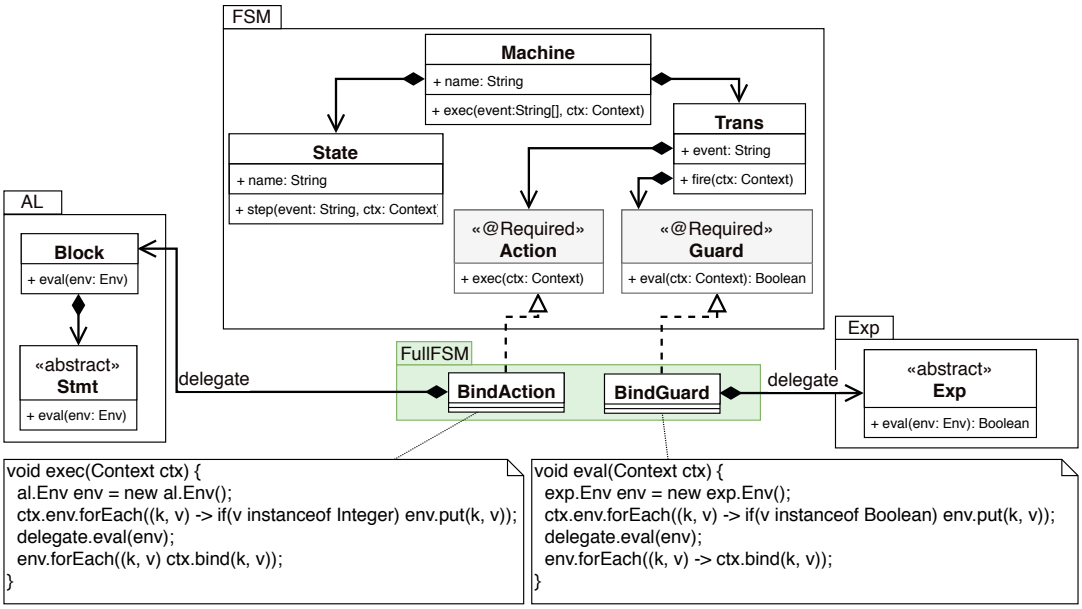

Figure 3. Composing the FSM language concern with an action language concern and an expression language concern. For the sake of conciseness, only excerpts of these concerns are depicted here

```
interface FullFSMEvalRev
  extends FullFSMRev< ... , EvalBindAction, ... >,
          FSMEvalRev,
          ALEvalRev,
          ExpEvalRev {
  default EvalBindAction bindAction(BindAction it) {
    return (ctx) → {
      al.Env env = new al.Env();
      ctx.env.forEach((k, v) →
        if(v instanceof Integer) env.put(k, v));
      delegate.eval(env);
      env.forEach((k, v) → ctx.bind(k, v));
    }
  }
  ...
}
```

Listing 5. Semantic implementation generated from the glue depicted in Figure 3

implementation of the semantics of the *Bind* meta-classes introduced in Section 4.2.1. The `FullFSMEvalRev` interface inherits from the Revisitor interface of the composed concern shown in Listing 4 and implements the execution functions of `BindAction` and `BindGuard`. These execution functions are the glue itself. For instance, the implementation of the factory method for `BindAction` is the implementation in Java of the glue depicted in Figure 3.

In conclusion, the composition of two language concerns is realized through syntactic bindings and semantic glue. This composition is implemented by a language concern itself and follows the exact same implementation pattern as that of a language concern. The semantic mappings, semantic interfaces, concrete semantic mappings, and semantic implementations, along with syntactic bindings and semantic glue, can all be written, type-checked, and compiled separately.

5 The Alex Language

Although the pattern introduced in Section 4 is intuitive and relies on well-known object-oriented concepts, it can be cumbersome and error-prone for language designers to implement their languages directly at the Java level, especially when the number of constructs in the concerns grows. To alleviate this problem, we present in this section Alex[2] (Action Language for Ecore with Xbase), a meta-language inspired by ALE [19] and Kermeta [15] dedicated to the implementation and composition of operational semantics.

Alex integrates seamlessly with the EMF ecosystem. In particular, it relies on Ecore for defining the abstract syntax of language concerns in the form of a metamodel, and allows to "re-open" meta-classes to define their operational semantics. The interoperability with EMF enables language designers using Ecore and Alex to benefit from other tools of the ecosystem, such as graphical editors implemented in Xtext or Sirius. In practice, Alex is an alternative to other approaches to semantic definition in the EMF ecosystem (Visitor pattern, EMF Switch, Xtend language, etc.), with a particular focus on modularity and composition.

Following the open class principle [6], Alex allows to "re-open" meta-classes of a metamodel to weave operational

[2] https://github.com/diverse-project/alex/

```
behavior evalfsm
import ecore "FSM.ecore"

open class Trans {
  def void fire(Context ctx) {
    if (!alg.$(obj.guard).eval(ctx))
      throw new RuntimeException("Unsatisfied guard");
    alg.$(obj.action).run(ctx)
    ctx.current = obj.tgt
  }
}
...
```

Listing 6. Firing a transition in Alex

semantics as a set of methods. Method bodies are written in Xbase [11], a simple yet powerful action language of the Xtext ecosystem that can be easily plugged and reused. Amongst other benefits, this choice allows us to reuse the tooling of Xbase including its built-in Java compiler.

As an illustrating example, Listing 6 depicts the definition, using Alex, of the `fire()` semantics of a transition for the FSM language depicted in Figure 1. The `open class` keyword re-opens the `Trans` meta-class, imported using the `import ecore` directive, to weave a new `fire` method into it. The `alg` and `obj` objects are readily available in the scope of Alex method bodies. The `alg` object refers to the current Revisitor interface through which the $-methods introduced in Section 4 can be invoked to retrieve executable semantic objects from syntactic objects. The `obj` object refers to a particular instance of the re-opened meta-class, a particular `Trans` object in this case. Composing multiple syntaxes and semantics is realized by using the `import alex` keyword. Writing the glue between two concerns involves importing the Ecore metamodel specifying the syntactic bindings between these two concerns, and importing their Alex specification. Then, the glue is implemented as methods in the corresponding *Bind* meta-classes.

Finally, from an Alex file, the Alex compiler automatically generates a Revisitor implementation following the pattern introduced in Section 4. Language designers thus focus on writing Ecore and Alex files; the generation of modular implementations in Java is fully automated and transparent.

6 Evaluation

This evaluation section is divided in three parts. First, Section 6.1 presents the IoT case study used to evaluate our approach. Section 6.2 details our implementation and analyzes it regarding the requirements of Section 2.3 in Section 6.2.1. Finally, Section 6.2.2 discusses the impact of our approach on metamodel complexity and run-time performance.

6.1 The IoT Case Study

To illustrate our approach, we re-implement a case study that was used to evaluate Melange in earlier work [9]. This case study consists in the definition of an executable modeling language for the Internet of Things (IoT) domain. It targets the definition of systems composed of multiple sensors and actuators deployed on resource-constrained micro-controller devices (e.g., Arduino, Raspberry Pi, etc.). This language is built by reusing various existing modeling languages and composing them to form the targeted IoT modeling language. We keep the same list of requirements, reminded below:

- the language must provide an Interface Definition Language (IDL) to model the sensor interfaces in terms of provided services;
- the language must support the modeling of concurrent sensor activities;
- the primitive actions that can be invoked within the activities must be expressed with a popular language IoT developers are familiar with.

To fulfill those requirements, Melange's case study selected respectively: the OMG's Interface Definition Language (IDL),[3] the UML Activity Diagram language extracted from the *Transformation Tool Context*,[4] and the scripting language Lua.[5] We reuse the same language concerns for our implementation.

Each language concern is implemented in its own Eclipse plug-in. Dependencies between the concerns are realized by the standard plug-in dependency mechanism offered by Eclipse.

6.2 Case Study Implementation

We implement the case study[6] by composing the three concerns detailed in Section 6.1, together with a fourth concern named `IoT` which introduces the domain-specific constructs of an IoT system.

Figure 4 depicts how those four concerns are composed together. The IoT concern includes two «`required`» constructs in its required interface: `IoTActivity` and `IoTOperationDef`. They are respectively bound to the `Activity` and `OperationDef` constructs of Activity Diagrams (AD) and IDL. The AD concern exposes three «`required`» constructs: `Expression`, `BooleanVariable` and `IntegerVariable`. The former is bound twice: to `OperationDef` of the IDL, and to `Statement` of Lua. The two other constructs are bound to the `Statement_Assignment` construct of Lua. Finally, the IDL concern exposes a «`required`» `IdlStmt` construct, bound to the `Block` construct of the Lua concern. This way, every «`required`» construct is bound to a concrete construct, and a fully defined language is formed.

[3] https://www.omg.org/spec/IDL/
[4] http://www.transformation-tool-contest.eu/
[5] https://www.lua.org/
[6] https://github.com/tdegueul/ale-xbase/tree/master/examples/composition/iot

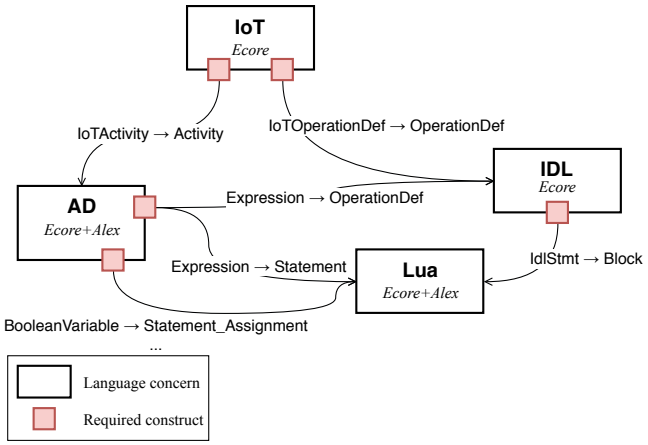

Figure 4. Definition of the IoT language as a composition of four language concerns

From these definitions, we use the built-in EMF compiler and our own custom compiler for Alex to derive Java implementations of the concerns themselves and their composition following the pattern guidelines presented in Section 4. Listing 7 presents an excerpt of the glue for the binding *Expression* → *OperationDef*.

The `ExpressionBindOperationDef` is defined in the metamodel of the IoT language. It inherits AD's `Expression` and has a field named `delegate` which references the `OperationDef` of IDL. In Listing 7, Alex is used to define the glue between `Expression` and `OperationDef` in the `execute` method of `ExpressionBindOperationDef`.

First, it initializes an `Environment` as expected by the IDL concern and populates it with the local variables of the AD concern. Then, it invokes the `execute` execution function of `OperationDef` through the $-method, passing the environment as argument. Finally, once the operation has been executed, it reads the values that have been possibly updated and translate them back in the AD context.

In the context of this case study, the glue between language concerns is mostly expressed at this level of abstraction and consists in the translation of variables and stores from one concern to the other.

6.2.1 Requirement Analysis

In this section, we discuss the requirements of Section 2 in the context of our case study.

Concern Encapsulation (R1) The definition of the glue is fully realized through inheritance and invocations of the semantic interfaces of language concerns. Hence, in order to compose the four language concerns that are part of our case study, we extend six classes, five methods are overridden for the definition of the glue, and two classes are needed to express how the internal evaluation contexts of

```
open class ExpressionBindOperationDef {
  override void execute(Context c) {
    // Initialize the OperationDef environment
    val e = new Environment
    c.inputValues.forEach[iv |
      e.putVariable(alg.$(iv.variable).name, iv.value)
    ]
    (obj.eContainer as OpaqueAction).activity.locals
    .forEach [l |
      e.putVariable(
        alg.$(l).name(),
        alg.$(l.currentValue).value
      )
    ]

    // Invoke the execution semantics of OperationDef
    alg.$(obj.delegate.stmt).execute(e)

    // Update the local context back
    obj.delegate.parameters.filter[p |
      #[ParameterMode::PARAM_OUT, ParameterMode::
     PARAM_INOUT]
      .contains(p.direction)
    ]
    .forEach [p |
      c.activity.locals.filter[l |
        alg.$(l).name() == p.identifier
      ]
      .forEach [l |
        alg.$(l.currentValue)
        .setValue(e.getVariable(p.identifier))
      ]
    ]
  }
}
```

Listing 7. Glue for the *Expression* → *OperationDef* binding in Alex

different concerns are translated. Language concerns without requirements do not have external dependencies towards other language concerns. Finally, the glue definitions only interact with a small and well-defined part of the reused language concerns. Those observations highlight the isolation capabilities of our approach.

Explicit Required Interface (R2) Concern requirements are easily identified by looking at the Ecore classes annotated with «`required`». While most of them are presented in Figure 4, the number of «`required`» classes per language concerns is: zero for Lua, one for the IDL, three for the AD, and two for the base IoT concern itself. Each of those «`required`» classes is bound exactly once except for the `Exp` class of the AD concern that is bound twice, one time to the IDL concern and another time to the Lua concern. Each «`required`» construct declares a single execution function, except for the `IoTOperationDef` construct that has no associated semantics.

This sums up the information needed for the composition and explicitly exposed in the language concerns interfaces.

Separate Compilation (R3) Each language is clearly isolated in its own Eclipse plug-in containing an Ecore model for its abstract syntax and an Alex file for its semantics. Each plug-in is type-checked and compiled separately. This means that the IoT language concern could be implemented by importing the Eclipse plug-ins of the other concerns from various places (including remotely, for instance) by using the standard Eclipse update site mechanism, which highlights the modularity of our approach. At the source code level, concerns interact with each others only though inheritance and reference to publicly exposed artifacts of the other concerns. As long as the interfaces of the concerns do not change, each concern can evolve internally without having to recompile other language concerns.

Concern Substitutability (R4) In our case study, as depicted in Figure 4, we bind two different expression language concerns to the AD concern: Lua and IDL can both be used to define expressions of the activity diagrams. From the point of view of the activity diagram concern, the choice of Lua or IDL is transparent. Using the glue between these concerns, Lua and IDL expressions can be used and evaluated indifferently.

Non-intrusivity (R5) The definition of the `«required»` interface is based only on the annotation of classes in the metamodel. This mechanism is built in EMF directly and does not require any intrusive change. The definition of modular languages leverages inheritance and the delegation pattern [13], which are well-known object-oriented concepts. The Revisitor implementation pattern is based on three object-oriented concepts: (i) parametric polymorphism (i.e., generics) with bounded type parameters (ii) multiple class or interface inheritance, and (iii) single dynamic dispatch. Such requirements are readily fulfilled by many mainstream object-oriented language (e.g., Java, C#) and their underlying runtime platforms (e.g., JVM, CLR). In conclusion, every part of our approach is based on existing and well-known object-oriented concepts, leading to a non-intrusive approach, which can be easily adapted to similar technological stacks.

6.2.2 Discussion

In addition to the requirements discussion, we now discuss two complementary aspects. What are the consequences of the introduction of intermediate *Bind* classes when composing languages through the delegation pattern, and what is the impact of modularity on the run-time performance of modular language interpreters?

The use of the delegation pattern, described in Section 4.2.1, leads to the introduction of new *Bind* meta-classes in the metamodels of composed concerns. These classes are needed to compose language concerns modularly but are merely technical artifacts that are not related to the domain constructs materialized by metamodels. Still, language designers must deal with them when adding new tools on top of the composed language concerns. In order to evaluate the cost of the introduction of these extra classes, we implemented a new Xtext grammar for the resulting IoT language. We observe that managing the extra *Bind* meta-classes requires to introduce additional intermediate production rules in the grammar. These new production rules are only needed to simulate delegation between the production rules of the composed language concerns. Such production rules are simple and do not require advanced grammar engineering knowledge. They account for 20 out of 555 lines in the grammar (<4%). We claim that this cost is largely compensated by the benefits of our approach regarding modularity and reuse.

In earlier work, we discussed the impact of the Revisitor pattern at run time [19]. As explained in Section 4, the definition of modular languages leads, at the implementation level, to multiple inheritance, delegation, and dispatch layers between the composed concerns. We already observed that, due to additional levels of dispatch that cannot be aggressively optimized by the JVM, the Revisitor pattern has a slight impact on performance, albeit reasonable. As the approach presented in this paper multiplies the number of dispatch, we expect the performance to be affected accordingly. Besides, the glue between language concerns (for instance, to translate local execution contexts from one concern to the other) may also affect performance negatively. A precise evaluation of this impact remains future work.

7 Related Work

Much work has been done on the definition of reusable languages. Nevertheless, none of them is fully integrated with mainstream (object-oriented) engineering technologies while supporting separate compilation.

Two approaches, Lisa [22] and Melange [9], provide support for language reuse within object-oriented frameworks and technologies. However, none of those approaches support separate compilation at the implementation level. Lisa can be distinguished from Melange by the technical space in which it evolves. Lisa is a grammar-first, attribute grammar meta-language while Melange is a model-first meta-language dedicated to the composition of metamodels and object-oriented operational semantics. Being model-first, our approach is influenced by Melange but improves over it by supporting separate compilation of language concerns.

Other approaches related to language composition enable separate compilation of languages but introduce new advanced paradigms and do not aim at being integrated into mainstream object-oriented technological stacks. Each of the following work pushed the boundaries of language concern composability in different technical spaces. Monticore [17], Neverlang [2], Rascal [1], and Spoofax [28] are grammar-first

language definition frameworks that allow the definition and composition of language; MPS [29] is a language workbench based on projectional editing. MontiCore introduces new advanced concepts in its grammar language (interfaces, aggregation, etc.) to support language composition. Neverlang offers fine-grained granularity for the composition of languages at the price of complex and specific concepts (slices, roles, etc.). Rascal and Spoofax, while not being explicitly dedicated to composability, have demonstrated their extensibility and composability. Both approaches are based on high-level, domain-specific, operators for the definition of software languages [1, 28]. In constrast, MPS is a projectional language workbench with favorable properties regarding language composability, mainly due to the absence of a parser. In comparison, our approach does not introduce ad-hoc concepts to allow the safe composition of language concerns and rely only on well known object-oriented concepts (i.e., inheritance, annotations, delegation).

At the implementation level, other approaches study the definition of advanced modular and type-safe interpreters. Zhang et al. share the use of code generation to abstract away from the engineering process generic parts of the interpreter pattern [30]. While we use an external DSL, Zhang et al. use Java's annotations to define a small internal DSL from which are generated the EVF visitors. They also explore the definition of modular general-purpose language semantics using EVF. Inostroza et al. [14] study the implicit propagation of execution context between modular interpreters. This work proposes solutions to decrease the needs for the implementation of error-prone glue code (e.g., Listing 7). Finally, a distinguishing property between those two works and the present paper is the absence of an abstract syntax in the form of an explicit metamodel, limiting the access to commonly available metamodel manipulation tools.

Besides, the question of the modularity of language concerns is strongly connected to the question of Software Product Line (SPL) and feature-oriented language development. Several works study languages from the point of view of software variability. Méndez-Acuña et al. [21] identify three approaches to language variability management [3, 18, 20]. The Concern-Oriented Language Development (COLD) approach [7] proposes the notion of language concern as the unit of software language reuse. By providing an explicit and modular language interface, our approach can contribute to the improvement of language families and feature-oriented programming solutions.

8 Conclusion and Future Work

In this paper, we have presented a modular implementation pattern for the definition and composition of language concerns. Language concerns are equipped with well-defined required interfaces that enable encapsulation and information hiding, and make explicit the requirements a concern has towards other concerns. Language concerns can be composed safely and modularly (i.e., with separate compilation and without anticipation), and without having to dive into their internal implementations. Our approach is integrated in Alex, an high-level object-oriented language dedicated to the definition of operational semantics on top of Ecore metamodels. We show that our approach is non-intrusive and modular through a case study consisting in the definition of a modular language for IoT systems modeling and simulation. This work is a first step towards a lightweight and efficient definition of software language product lines. It would be of great interest to study how our pattern can help achieve the vision of modular families of software languages.

Acknowledgment

This work is partially supported by the CWI/Inria associate team ALE (http://gemoc.org/ale/), the EU's Horizon 2020 Project No. 732223 CROSSMINER, and by the Direction Générale de l'Armement (Pôle de Recherche CYBER). Also, we would like to thank the anonymous reviewers for their helpful comments.

References

[1] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen J. Vinju. 2015. Modular language implementation in Rascal - experience report. *Sci. Comput. Program.* 114 (2015), 7–19. https://doi.org/10.1016/j.scico.2015.11.003

[2] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2 - Componentised Language Development for the JVM. In *Software Composition - 12th International Conference, SC 2013, Budapest, Hungary, June 19, 2013. Proceedings (Lecture Notes in Computer Science)*, Walter Binder, Eric Bodden, and Welf Löwe (Eds.), Vol. 8088. Springer, 17–32. https://doi.org/10.1007/978-3-642-39614-4_2

[3] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings (Lecture Notes in Computer Science)*, Andy Schürr and Bran Selic (Eds.), Vol. 5795. Springer, 670–684. https://doi.org/10.1007/978-3-642-04425-0_54

[4] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications. *Trans. Aspect-Oriented Software Development* 12 (2015), 132–179. https://doi.org/10.1007/978-3-662-46734-3_4

[5] Tony Clark. 1999. Type Checking UML Static Diagrams. In *«UML»'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings (Lecture Notes in Computer Science)*, Robert B. France and Bernhard Rumpe (Eds.), Vol. 1723. Springer, 503–517. https://doi.org/10.1007/3-540-46852-8_36

[6] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. 2000. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000.*, Mary Beth Rosson and Doug Lea (Eds.). ACM, 130–145. https://doi.org/10.1145/353171.353181

[7] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule,

Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. 2018. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems and Structures* (2018), 1–26. https://doi.org/10.1016/j.cl.2018.05.004

[8] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science)*, James Noble (Ed.), Vol. 7313. Springer, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

[9] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a meta-language for modular and reusable development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, Richard F. Paige, Davide Di Ruscio, and Markus Völter (Eds.). ACM, 25–36. https://doi.org/10.1145/2814251.2814252

[10] Thomas Degueule, Benoît Combemale, and Jean-Marc Jézéquel. 2017. On Language Interfaces. In *Present and Ulterior Software Engineering.*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer, 65–75. https://doi.org/10.1007/978-3-319-67425-4_5

[11] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. 2012. Xbase: implementing domain-specific languages for Java. In *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, Klaus Ostermann and Walter Binder (Eds.). ACM, 112–121. https://doi.org/10.1145/2371401.2371419

[12] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. 2010. Empirical Language Analysis in Software Linguistics. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, Brian A. Malloy, Steffen Staab, and Mark van den Brand (Eds.), Vol. 6563. Springer, 316–326. https://doi.org/10.1007/978-3-642-19440-5_21

[13] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software.* Pearson Education India.

[14] Pablo Inostroza and Tijs van der Storm. 2017. Modular interpreters with implicit context propagation. *Computer Languages, Systems & Structures* 48 (2017), 39–67. https://doi.org/10.1016/j.cl.2016.08.001

[15] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. 2015. Mashup of metalanguages and its implementation in the Kermeta language workbench. *Software and System Modeling* 14, 2 (2015), 905–920. https://doi.org/10.1007/s10270-013-0354-4

[16] Lennart C. L. Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 444–463. https://doi.org/10.1145/1869459.1869497

[17] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2014. MontiCore: Modular Development of Textual Domain Specific Languages. *CoRR* abs/1409.6633 (2014). arXiv:1409.6633 http://arxiv.org/abs/1409.6633

[18] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and picky: configuration of language product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, Douglas C. Schmidt (Ed.). ACM, 71–80. https://doi.org/10.1145/2791060.2791092

[19] Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs van der Storm, and Olivier Barais. 2017. Revisiting Visitors for Modular Extension of Executable DSMLs. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*. IEEE Computer Society, 112–122. https://doi.org/10.1109/MODELS.2017.23

[20] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-oriented language families: a case study. In *The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013*, Stefania Gnesi, Philippe Collet, and Klaus Schmid (Eds.). ACM, 11:1–11:8. https://doi.org/10.1145/2430502.2430518

[21] David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. *Computer Languages, Systems & Structures* 46 (2016), 206–235. https://doi.org/10.1016/j.cl.2016.09.004

[22] Marjan Mernik. 2013. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software* 86, 9 (2013), 2451–2464. https://doi.org/10.1016/j.jss.2013.04.087

[23] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. 2005. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings (Lecture Notes in Computer Science)*, Lionel C. Briand and Clay Williams (Eds.), Vol. 3713. Springer, 264–278. https://doi.org/10.1007/11557432_19

[24] OMG. 2006. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/spec/MOF/2.0/. (2006).

[25] Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. 2015. Pattern-based development of Domain-Specific Modelling Languages. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, Timothy Lethbridge, Jordi Cabot, and Alexander Egyed (Eds.). IEEE Computer Society, 166–175. https://doi.org/10.1109/MODELS.2015.7338247

[26] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework.* Pearson Education.

[27] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40. https://doi.org/10.1016/j.cl.2015.02.001

[28] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz (Eds.). ACM, 95–111. https://doi.org/10.1145/2661136.2661149

[29] Markus Voelter. 2011. Language and IDE Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers (Lecture Notes in Computer Science)*, Ralf Lämmel, João Saraiva, and Joost Visser (Eds.), Vol. 7680. Springer, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11

[30] Weixin Zhang and Bruno C. d. S. Oliveira. 2017. EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 29:1–29:32. https://doi.org/10.4230/LIPIcs.ECOOP.2017.29